RELOGO GETTING STARTED GUIDE

JONATHAN OZIK - REPAST DEVELOPMENT TEAM

0. Before we Get Started

Before we can do anything with Repast Simphony, we need to make sure that we have a proper installation of Repast Simphony 2.1. Instructions on downloading and installing Repast Simphony on various platforms can be found on the Repast website[1]. Repast Simphony 2.1 requires Java 7. Java 7 can be found at the Java Standard Edition Downloads page[2].

1. Getting Started with ReLogo

Now let us begin our exploration of ReLogo. We will be building a simple agent-based model involving zombies chasing humans and humans running away from zombies. Our approach will be to not overwhelm you but to explain only as much as is needed at each step. By the end of this chapter, we'll have covered a lot of ground and you'll be able to continue with your own explorations of ReLogo[3].

The first thing we must do is create a new ReLogo project. This is done by clicking on the New ReLogo Project icon in the toolbar (Fig. 1) at the top of our ReLogo workspace.

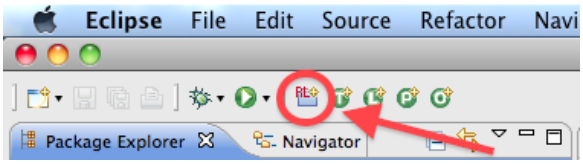

Figure 1. The New ReLogo Project icon.

This brings up the New ReLogo Project Wizard (Fig. 2) which gives us the ability to name our project (and a few more options which we'll ignore for now). Typing in

Date: July 18, 2013.

[1]http://repast.sourceforge.net/download.html

[2]http://www.oracle.com/technetwork/java/javase/downloads/index.html

[3]For further reading see Ozik, J., N. Collier, J. Murphy, and M.J. North. "The ReLogo Agent-based Modeling Language." In WSC 2013 Proceedings. Washington, D.C., December 2013.

"Zombies" in the "Project name" field[4], we press the "Finish" button and the wizard sets up our project. The project structure should look like Fig. 3 (but if it does not, see this note[5]).

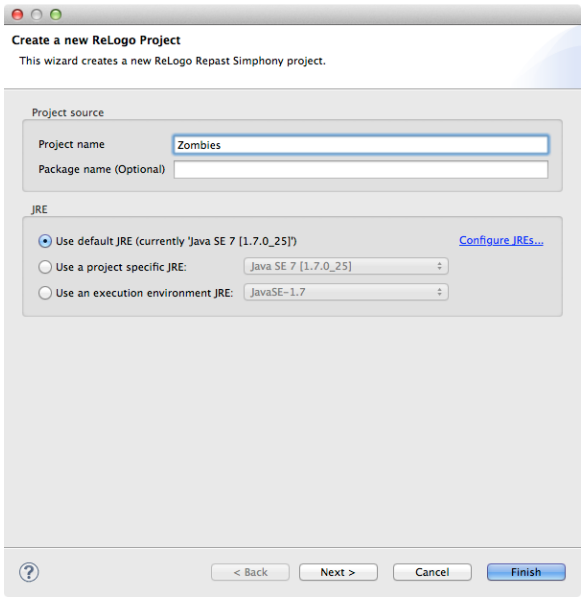

FIGURE 2. The New ReLogo Project Wizard.

What we see is the Zombies project folder, the "src" subfolder, the "zombies.relogo" package, and the relevant ReLogo files, in addition to a "shapes" folder. More on each of these things as we proceed.

1.1. **Creating the Human and Zombie turtle types.** Now that we have all our model infrastructure in place, we can start specifying the details of our Zombie model. First, since zombies love to chase humans, we create the Human turtle[6] type[7]. We do this by selecting the "zombies.relogo" package, if it isn't selected, and then clicking on the New Turtle icon (Fig. 4) in the toolbar[8]. This brings up the New Turtle Wizard (Fig. 5) which allows us to specify the name of our turtle type (Human). If we initially selected the "zombies.relogo"

[4]ReLogo differentiates between capitalized and uncapitalized letters (i.e., it is case sensitive) so, when following this tutorial, it will be important to note the capitalization of names, variables, etc.

[5]Here we assume that your ReLogo Resource Filter is enabled. If the ReLogo Resource Filter is disabled, you may see more elements in your Zombies project. See Section 2.4 on how to disable/enable this filter.

[6]In Logo dialects a "turtle" is a mobile agent.

[7]Many of the ReLogo entities we'll encounter, including turtle types, are what are known in Object Oriented programming languages as *classes*.

[8]Selecting the "zombies.relogo" package simplifies the next step.

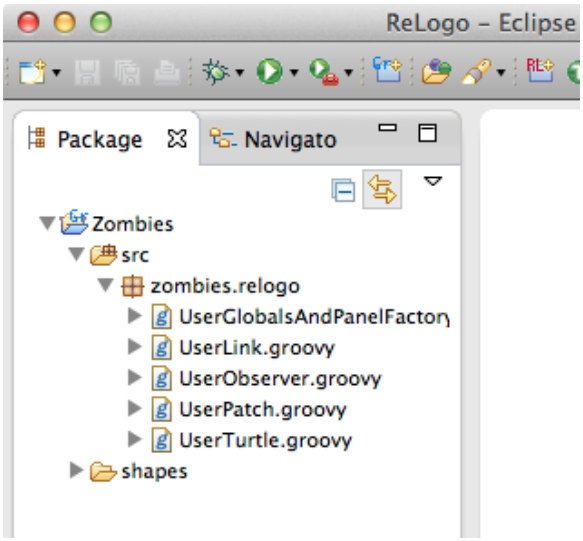

FIGURE 3. The directory structure of the newly created Zombies project.

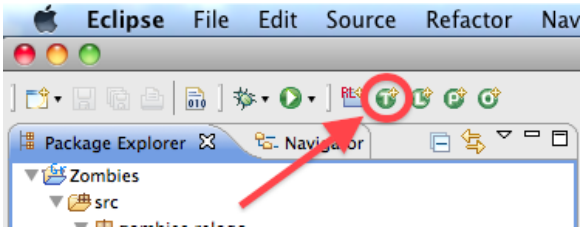

FIGURE 4. The New Turtle icon.

package, we simply fill in the Name field with "Human" and hit the Finish button[9]. At this point we should be greeted by our newly created Human turtle type as seen in Fig. 6[10]

Next we follow a similar procedure to create the Zombie turtle type (Fig. 7).

[9]If the "zombies.relogo" package hadn't been selected, we need to fill in the Package field with "zombies.relogo" before hitting the Finish button.

[10]For those curious about the .groovy file ending, ReLogo is an agent-based modeling domain specific language (ABM DSL), written in the Groovy programming language. While it isn't necessary to be able to follow this getting started guide, we recommend getting to know the language through the Groovy website (http://groovy.codehaus.org) and the many Groovy books which are available. There is also a very convenient web based console (http://groovyconsole.appspot.com/) where you can experiment with the Groovy language.

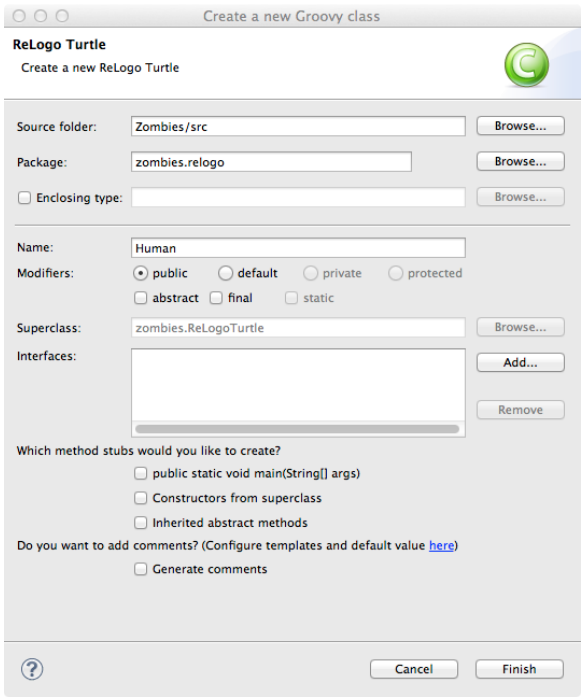

FIGURE 5. The New Turtle Wizard with information for creating the Human turtle type.

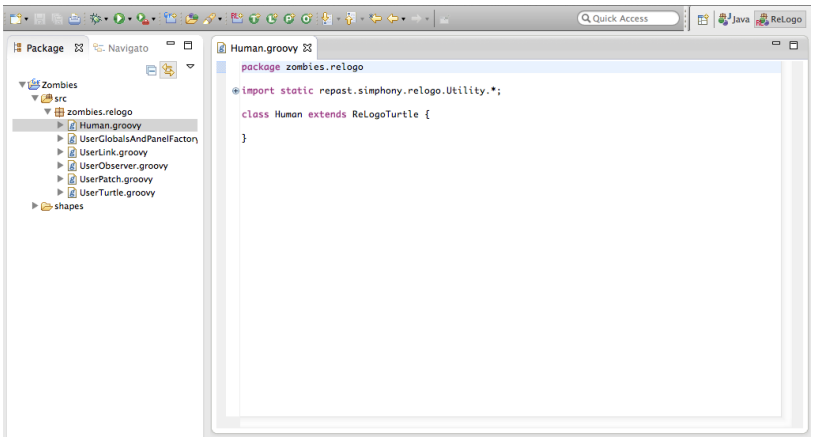

FIGURE 6. The view after creation of the Human turtle type.

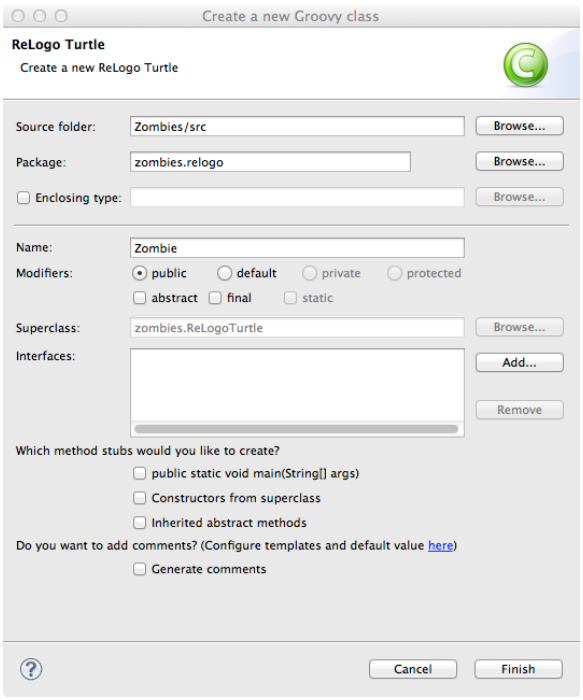

FIGURE 7. The New Turtle Wizard with information for creating the Zombie turtle type.

1.2. **Defining Human and Zombie behaviors.** The next step in building our model is defining the behaviors of the Human and Zombie turtle types. We'll dive right in.

Each Human will have a *step* method which looks like this (but first, see important note[11]):

```
1  def step(){
2     def winner = minOneOf(neighbors()){
3        count(zombiesOn(it))
4     }
5     face(winner)
6     forward(1.5)
7     if (infected){
8        infectionTime++
9        if (infectionTime >= 5){
10          hatchZombies(1){
11             size = 2
12          }
13          die()
14       }
15    }
16 }
```

Listing 1. Human step method.

The thinking here is that at every time advancement of our simulation, or "tick," each Human will execute this method. The Human first chooses a *winner* which is, in plain English, one of the neighboring patches[12] with the fewest number of Zombies on it. When the winner is chosen, the Human *face*s it and moves *forward* 1.5 steps, thereby running away from potential high Zombie areas. If the Human is *infected*, and 5 or more time ticks have passed since the initial infection, the Human *die*s and *hatch*es a Zombie.

Let's briefly review the code. A ReLogo turtle has a number of ReLogo primitives[13], or capabilities, it can use, without having to create its own. Among these are *minOneOf* and *neighbors*. *minOneOf* takes as an argument a set of "things" and a block defining

[11]For those following this getting started guide electronically, simple copying-and-pasting of the code from the tutorial will result in errors. Even if the line numbers are removed there can be errors resulting from the formatting of quoted strings and other elements. In short, to get the most out of the guide with the least amount of errors, we recommend typing in the code yourself.

[12]Patches were introduced by StarLogo and are square shaped immobile agents which make up an underlying grid structure in a ReLogo world.

[13]The set of available primitives can be found in the ReLogoPrimitives.html file that came with the Repast Simphony distribution. The primitives are separated broadly by the type of ReLogo entity that uses them and further by the type of primitive category (e.g, motion, rotation, etc.). Clicking on a primitive will give you more details. In addition to the information available within the ReLogo editor itself (see Fig. 8), this will be an additional reference as you explore ReLogo so it will help to get familiar with it.

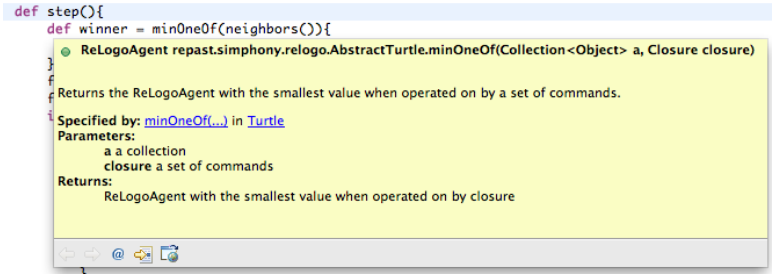

FIGURE 8. Hovering over any element in the editor will reveal an informational pane. In this case the information related to the *minOneOf* primitive is shown.

what quantity will be used to determine the minimum of. Hovering over the *minOneOf* primitive reveals an information pane describing it (Fig. 8).

The "things" (in line 2 of Listing 1) are the set of patches returned by the *neighbors* primitive, which are the 8 neighbors[14] of the patch that the Human is on. You'll notice that *neighbors* is followed by a set of empty parentheses. This is because the primitives are what are referred to in some programming circles as *methods* and the parentheses hold the arguments to those *methods*. If the method doesn't take any arguments, as is the case with the *neighbors* primitive, we still need to call the method but without any arguments. As mentioned above, *minOneOf* takes two arguments, the second being a block of code, specified between curly braces. Groovy allows, for clarity, to omit the parentheses around a block of code if it's the last argument to a method. So instead of writing:

```
maxOneOf(neighbors() , {
    count(zombiesOn(it))
} )
```

we can write:

```
maxOneOf(neighbors()){
    count(zombiesOn(it))
}
```

The block of code is executed in the context of the first argument to *minOneOf*, in this case the 8 neighboring patches.

[14]This is referred to as the Moore neighborhood in a grid.

When you define a turtle type in ReLogo, there are primitives that are automatically made available to various ReLogo entities[15]. *zombiesOn* is such a primitive that was made available to patches when we defined the Zombie turtle type. It takes as an argument a patch[16] and returns the Zombie turtles on that patch. The *it* is another feature of Groovy. It is an implicit argument to the block of code[17] which allows us to simplify:

```
maxOneOf(neighbors()){ p ->
   count(zombiesOn(p))
}
```

to:

```
maxOneOf(neighbors()){
   count(zombiesOn(it))
}
```

The primitive *count* can be applied to any set of items and returns the number of items in the set. Finally, the *def* keyword in Groovy is used when we define variables or the return types of methods. It's basically a wildcard indicating that whatever it adorns is of "some" type, without specifying further[18]. Thus, taken together, lines 2-4 of Listing 1, *assign* to the variable *winner* the patch with the fewest number of Zombies on it[19].

We'll assume that lines 5 and 6 of Listing 1 are self explanatory, and proceed to the *conditional* statement on lines 7-13. The *if* keyword is commonly used in many programming languages to determine the logical flow of some statements. In this case, we are checking to see if the Human is *infected* and if so, we proceed to line 8 and otherwise we skip down past line 13, the end of the *if* block. This is a good time to introduce the fact that in addition to methods, ReLogo entities have properties as well. Thus, elsewhere (which we'll show in Listing 2), we've explicitly specified that the Human turtle type has an *infected* property which we will initially set to *false*. The same holds for the *infectionTime* property[20], on line 8, which we'll initially set to 0, again elsewhere (Listing 2).

---

[15]For a complete list, see Table 1 in Appendix A.

[16]*zombiesOn* can also take a turtle as an argument, which has the same semantics as the version that takes a patch as an argument, but in the former case the patch is the patch under the turtle.

[17]Arguments passed to a block of code are a comma separated sequence of variable names followed by `->`. For any block of code without input arguments explicitly specified, the assumption is that it takes one implicit parameter named *it*.

[18]Groovy, for those interested, uses dynamic (but strict) typing. This is just one of the reasons why we often refer to it as a "less neurotic" Java.

[19]One of the most common mistakes is replacing the equality operator `==` with the assignment operator `=`. In the former case, the *equality* of the two sides is checked and either a *true* or *false* is returned. In the latter case, the right hand side is *assigned* to the left hand side.

[20]The `infectionTime++` notation is a shortcut for `infectionTime = infectionTime + 1`.

On lines 9-12, we check to see if the *infectionTime* is greater than or equal to 5 and, if so, the Human *hatches* a Zombie and then it *dies*[21].

Now let's see what our full Human turtle type looks like, with the *step* method and the turtle properties:

```
1  // package declaration and imports, which we can ignore for now
2
3  class Human extends BaseTurtle {
4
5     def infected = false
6     def infectionTime = 0
7
8     def step(){
9        def winner = minOneOf(neighbors()){
10          count(zombiesOn(it))
11       }
12       face(winner)
13       forward(1.5)
14
15       if (infected){
16          infectionTime++
17          if (infectionTime >= 5){
18             hatchZombies(1){
19                size = 2
20             }
21             die()
22          }
23       }
24    }
25
26 }
```

Listing 2. The Human turtle type.

The takeaway here is that turtle type properties and methods are defined within the *class* body of the turtle type, in this case between the curly braces on lines 3 and 26.

Now let's move on to the Zombie turtle type. It looks like this:

[21]The order might be confusing but if the Human dies and is removed from the simulation before hatching the Zombie, well, it can't hatch the Zombie!

```
1  // package declaration and imports, which we can ignore for now
2
3  class Zombie extends BaseTurtle {
4
5     def step(){
6        def winner = maxOneOf(neighbors()){
7           count(humansOn(it))
8        }
9
10       face(winner)
11       forward(0.5)
12
13       if (count(humansHere()) > 0){
14          label = "Brains!"
15          infect(oneOf(humansHere()))
16       }
17       else {
18          label = ""
19       }
20    }
21
22    def infect(human){
23       human.infected = true
24    }
25 }
```

LISTING 3. The Zombie turtle type.

What should be noticed here is that, in addition to the method *step*, we've defined another auxiliary method *infect* which takes one argument. Let us take a moment to explore how it's used. Looking at lines 13-15 in Listing 3, we see that we check using the primitives *count* and *humansHere*[22] if there are any Human turtle types "here." If there are, we define the default turtle type property *label* to be "Brains!" and proceed to *infect oneOf* the *humansHere*[23]. To infect the human, we access the human's property *infected* by *referencing* it with a period[24].

[22]As you probably guessed, this is one of the generated primitives when we defined the Human turtle type.

[23]As you likely notice, the purpose of Logo constructs in general and ReLogo code in particular can often easily be understood and can lead to more manageable code.

[24]In Object Oriented languages, accessing an object's properties and methods is a common idiom.

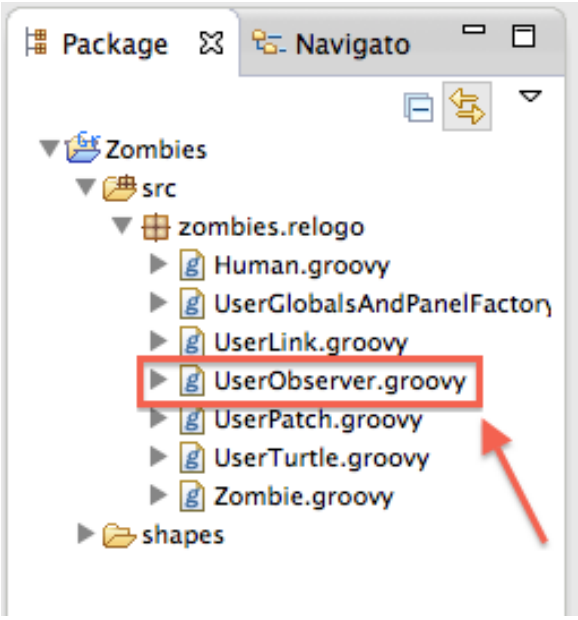

FIGURE 9. The UserObserver.groovy file in the Package Explorer view.

1.3. **Coordinating behaviors with the UserObserver.** At this point we have both the Human and Zombie turtle types specified. What we need next is to define the overall flow of our model. Logo uses the notion of an "observer" to accomplish this role. The UserObserver is the default observer type that was made for us when we created the Zombies project. Double clicking the UserObserver.groovy file in the Package Explorer view (Fig. 9) will bring up the file[25].

A common idiom is to define a method that sets up our simulation. We do so by defining the *setup* method:

[25]When the UserObserver.groovy file is revealed in the editor it will include *commented* sections of code. Comments in ReLogo and Groovy (and Java) are specified by `//` for single line comments and `/*comment content*/` for multiline comments. All the contents of comments are ignored in terms of program logic but they are helpful (and many say indispensable) for creating readable and manageable code. You may leave the comments in or delete them.

```
1  @Setup
2  def setup(){
3     clearAll()
4     setDefaultShape(Human, "person")
5     createHumans(numHumans){
6        setxy(randomXcor(),randomYcor())
7     }
8     setDefaultShape(Zombie, "zombie")
9     createZombies(numZombies){
10       setxy(randomXcor(),randomYcor())
11       size = 2
12    }
13 }
```

LISTING 4. The UserObserver setup method.

We'll first describe, in plain English, what this piece of code does. It is a good idea in any initialization code to reset the simulation, so we first *clearAll* existing entities from the world. Then we define the default shapes of both the Humans and Zombies and create a different number of each (with primitives generated by defining the Human and Zombie turtle types). We also scatter the created turtles randomly across our world and set the size of the Zombies to 2, to make them more prominent.

Now that we know the gist of Listing 4, let's delve a little deeper. Our method is *annotated* with the `@Setup` annotation. This indicates to the system that this method should be run when the system schedule starts up[26]. The *clearAll* primitive removes all existing entities from the ReLogo world and resets the patches to their default state. The *setDefaultShape* primitive takes two arguments. The first argument is a turtle type and the second is a *string*[27] specifying a turtle shape. To understand what shapes are available we can open the "shapes" folder in the Package Explorer view (Fig. 10) and see the default shapes that come with any newly created ReLogo project[28]. Simply specifying the name of one of these shapes (without the .svg suffix) in the *setDefaultShape* primitive's second argument will set the chosen turtle type's default shape to it[29].

Another thing that the astute reader might notice is that there are references to *numHumans* and *numZombies*. While it's possible for these to be properties of the UserObserver

[26]More specifically, this schedules the *setup* method to run at simulation *tick* 0. Naming our method *setup* was done for clarity but it could have been named anything. The important point is that `@Setup` annotates the method we'd like to use for our model setup.

[27]A string is a set of characters enclosed in quotes.

[28]In Appendix C we'll see that you can import a variety of shape types and even create your own.

[29]This just sets the default shape of the turtle type. Shapes can be subsequently changed during the simulation as well with the shape = "shapeName" construct.

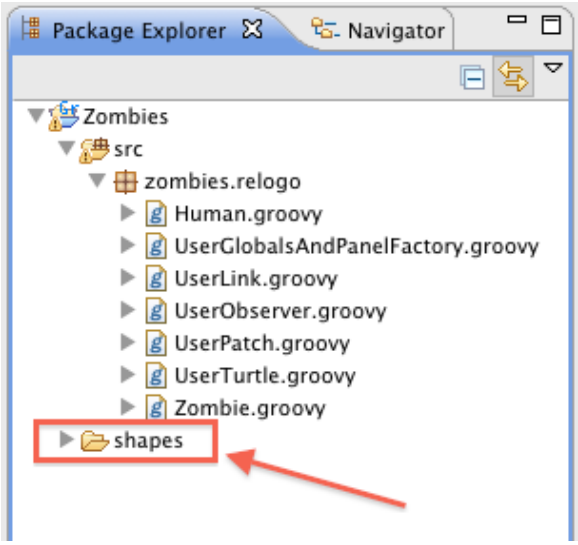

FIGURE 10. The shapes folder in the Package Explorer view.

class, we will opt to make these "tweakable" as the simulation progresses via slider graphical elements. More on this in a bit. The ReLogo editor has autocompletion capabilities. Starting to type `createHu` (on Line 5 in Listing 4) and pressing Control Space will reveal suggested completions (Fig. 11). The information in the revealed pane indicates that the *createHumans* method (and the *createZombies* method) takes an optional block of code as a second argument where the created turtles can be initialized immediately after being created. The contents of the code block are understood to be in the context of the relevant entities (Human or Zombie turtles) so it is unnecessary to specify that it's the turtle `setxy` method that we are calling. Invoking the code completion within the code blocks illustrates this (Fig. 12).

The next thing for us to do is to define the simulation step. That is, to coordinate what happens with the Zombie and Human turtles as time advances. We do this in our *go* method.

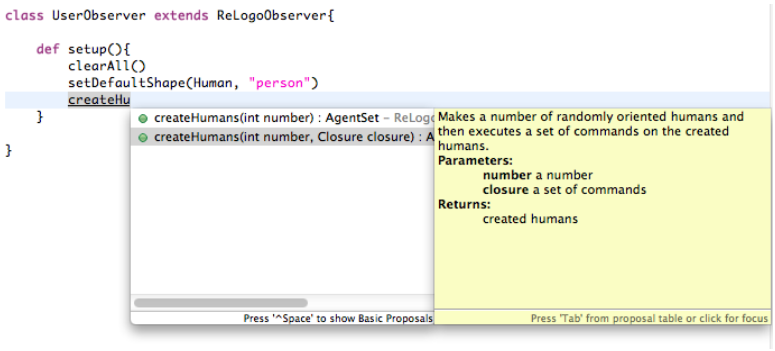

FIGURE 11. Autocompletion in the ReLogo editor. Pressing Control Space will reveal suggestions for completions.

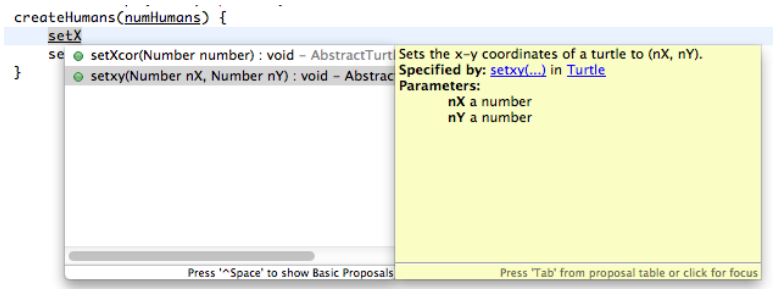

FIGURE 12. Autocompletion within the *createHumans* optional code block. The editor understands that the contents of the code block are in the context of turtles (as opposed to just the observer context) by the suggestions `setXcor` and `setxy`.

```
1 @Go
2 def go(){
3     ask (zombies()){
4         step()
5     }
6     ask (humans()){
7         step()
8     }
9 }
```

LISTING 5. The UserObserver go method.

As we see, we've already done most of the hard work of specifying the agents' behaviors.

The `@Go` annotation tells the system to run the *go* method starting at time 1 (in units of simulation *ticks*), repeating every subsequent *tick*[30]. What the observer does is *ask* the Zombie and Human turtle types to execute their *step*. This is achieved with the Logo idiom *ask*. A ReLogo entity (observer, turtle, patch, etc.) can *ask* another entity or set of entities to do things by specifying the request in a block of code after the *ask* primitive[31]. In this case, the *zombies* and *humans* primitives[32] return the set of all Zombie and Human turtle types respectively. The context of the code block passed to *ask* is that of the *ask*ed entities.

Collecting what we've covered so far in the observer code, the UserObserver class should look like this:

[30]Here, once again, we could have named the method something other than *go*.

[31]The block of code is the second argument to the *ask* primitive.

[32]Some more examples of automatically generated primitives.

```
1  // package declaration and imports, which we can ignore for now
2
3  class UserObserver extends BaseObserver{
4
5     /**
6      * Some comments here.
7      */
8     @Setup
9     def setup(){
10       clearAll()
11       setDefaultShape(Human, "person")
12       createHumans(numHumans){
13          setxy(randomXcor(),randomYcor())
14       }
15       setDefaultShape(Zombie, "zombie")
16       createZombies(numZombies){
17          setxy(randomXcor(),randomYcor())
18          size = 2
19       }
20
21    }
22
23    @Go
24    def go(){
25       ask (zombies()){
26          step()
27       }
28       ask (humans()){
29          step()
30       }
31    }
32
33 }
```

LISTING 6. The UserObserver class.

1.4. **Creating the graphical control and display elements.** We've specified the agent (turtle) behaviors and the overall flow of our Zombies model. Here we add some controls via graphical elements. The relevant file is UserGlobalsAndPanelFactory.groovy (Fig. 13).

Opening this file we immediately notice some, hopefully helpful, comments on what types of elements are available. For a complete list, see Appendix B. We create slider elements,

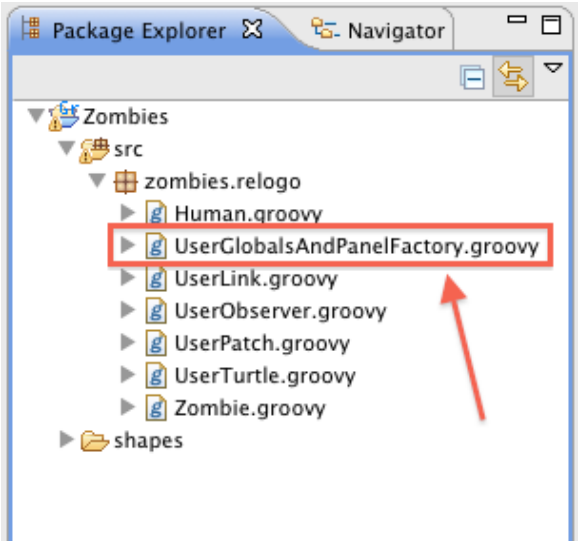

FIGURE 13. The UserGlobalsAndPanelFactory.groovy file in the Package Explorer view.

with labels, for the *numHumans* and *numZombies* variables referenced in UserObserver (Listing 6)[33]. Sliders allow one to vary the value of a variable by dragging.

```
addSliderWL("numHumans", "Number of Humans", 1, 1, 100, 50)
addSliderWL("numZombies", "Number of Zombies", 1, 1, 10, 5)
```

Let's anticipate that we'd like to know, at any given time, the number of remaining Humans. We employ a monitor for this.

```
addMonitorWL("remainingHumans", "Remaining Humans", 5)
```

A monitor takes as its first argument the name (as a string) of a method in our UserObserver, which we'll have to create below. The second argument is the label of the monitor and the third argument specifies how often we'll be updating the monitor, in this case every 5 time ticks.

[33]A trailing `WL`, i.e. "with label", is added to the graphical element methods to signify that the method takes a label as an argument. Otherwise the name of the relevant object (in the case of a slider, the variable referred to by the slider) will be displayed.

Our UserGlobalsAndPanelFactory should look like[34]:

```
1  // package declaration and imports, which we can ignore for now
2
3  public class UserGlobalsAndPanelFactory
4  extends AbstractReLogoGlobalsAndPanelFactory{
5
6     public void addGlobalsAndPanelComponents(){
7
8        /**
9         * Example comments
10        */
11
12       addSliderWL("numHumans", "Number of Humans", 1, 1, 100, 50)
13       addSliderWL("numZombies", "Number of Zombies", 1, 1, 10, 5)
14       addMonitorWL("remainingHumans", "Remaining Humans", 5)
15    }
16
17 }
```

LISTING 7. The UserGlobalsAndPanelFactory class.

The *remainingHumans* method to be added to the UserObserver class is as follows [35]:

```
1  def remainingHumans(){
2     count(humans())
3  }
```

LISTING 8. The *remainingHumans* method in UserObserver.

The UserObserver with the additional *remainingHumans* method should now look like this:

[34]Note that all the graphical elements are specified *within* the curly braces after *addGlobalsAndPanelComponents()*, in this case on lines 7 - 14.

[35]As a note, in Groovy the use of the `return` keyword to return a value from a method is optional when the returned value is given by the last statement in the method. This is because the value of the last statement in a method is automatically returned to the method's caller. Thus we can write `count(humans())` instead of `return count(humans())` in Listing 8.

```
1  // package declaration and imports, which we can ignore for now
2
3  class UserObserver extends BaseObserver{
4
5     /**
6      * Some comments here.
7      */
8     @Setup
9     def setup(){
10       clearAll()
11       setDefaultShape(Human, "person")
12       createHumans(numHumans){
13          setxy(randomXcor(),randomYcor())
14       }
15       setDefaultShape(Zombie, "zombie")
16       createZombies(numZombies){
17          setxy(randomXcor(),randomYcor())
18          size = 2
19       }
20
21    }
22
23    @Go
24    def go(){
25       ask (zombies()){
26          step()
27       }
28       ask (humans()){
29          step()
30       }
31    }
32
33    def remainingHumans(){
34       count(humans())
35    }
36
37
38 }
```

LISTING 9. The UserObserver class.

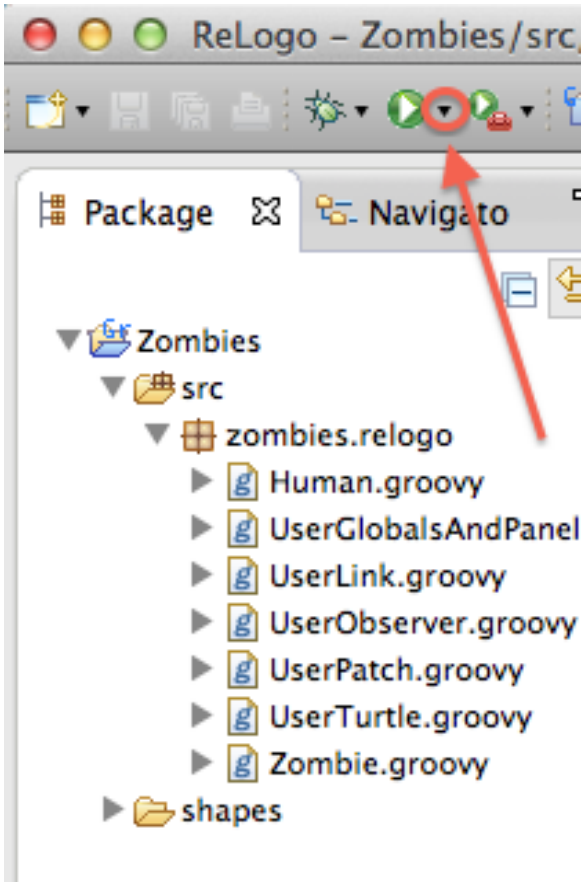

FIGURE 14. The downward triangle to reveal the Run options pull down menu.

1.5. **Running the Zombies model.** Our next step is to see what happens when we actually run the Zombies model that we've built so far. We do this by clicking on the small downward triangle to the right of the green "play" button in the toolbar at the top of the ReLogo workspace (Fig. 14). This reveals a pull down menu and we select the "Zombies Model" choice (Fig. 15).

This launches the Repast Simphony runtime (Fig. 16). The next step is to press the *Initialize* button to initialize the runtime (Fig. 17) which loads our Zombie model (Fig. 18).

The User Panel on the left reveals the graphical elements that we created (Fig. 19). Pressing the *Step* button at the top of the runtime (Fig. 20) will advance the simulation schedule to time tick 0 and will, therefore, run the *setup* method we scheduled with the `@Setup` annotation in the UserObserver (Listing 9). This should result in Figure 21.

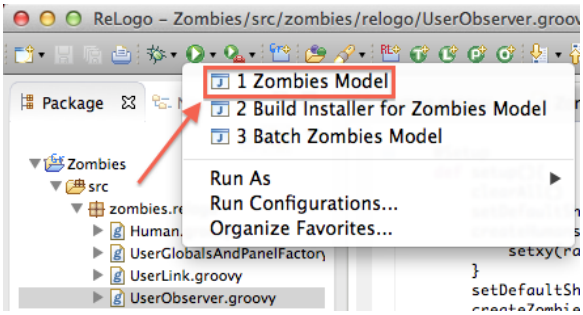

FIGURE 15. The "Zombies Model" entry which launches the Zombies model.

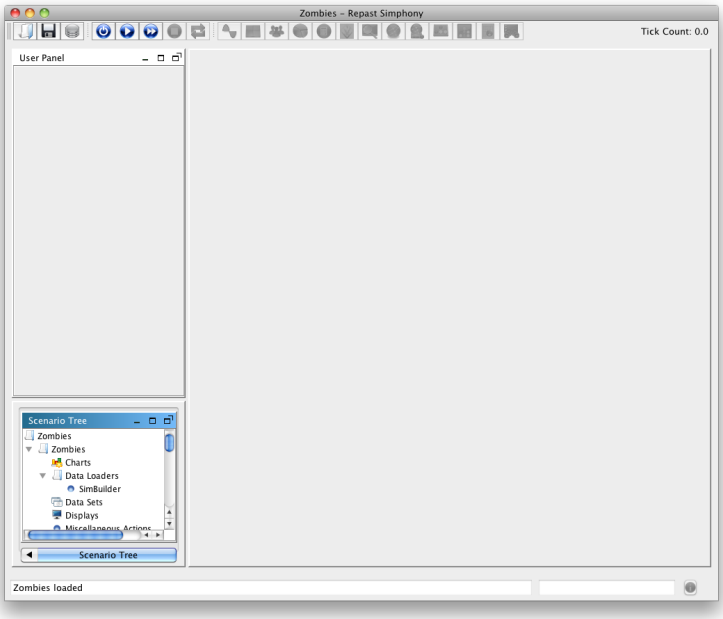

FIGURE 16. The Repast Simphony runtime.

At this point we can choose to repeatedly press the *Step* button and observe how the model evolves (Fig. 22) or, alternatively, we can press the *Play* button (Fig. 23) which will run the simulation until we *Pause* (Fig. 24) or *Stop* (Fig. 25) it. Whenever we want to reset the model, we simply press the *Reset* (Fig. 26) button.

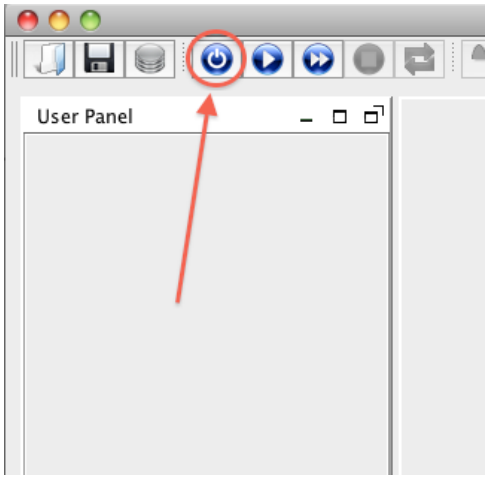

FIGURE 17. The Initialize button in the Repast Simphony runtime.

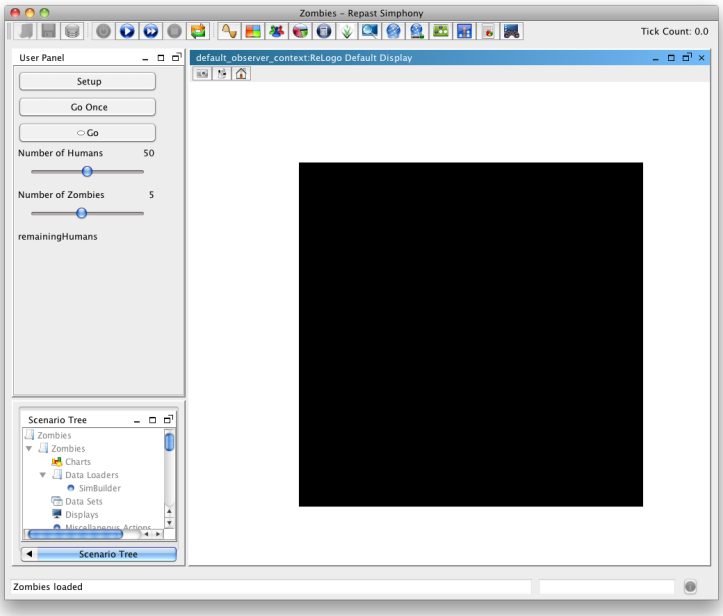

FIGURE 18. The loaded Zombie model.

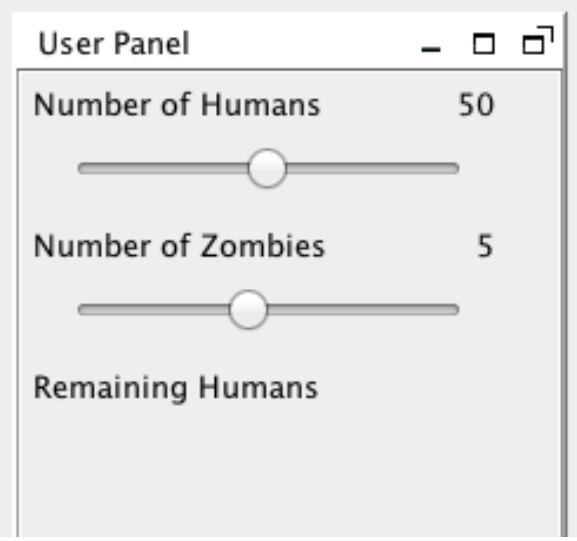

FIGURE 19. The graphical elements in the User Panel.

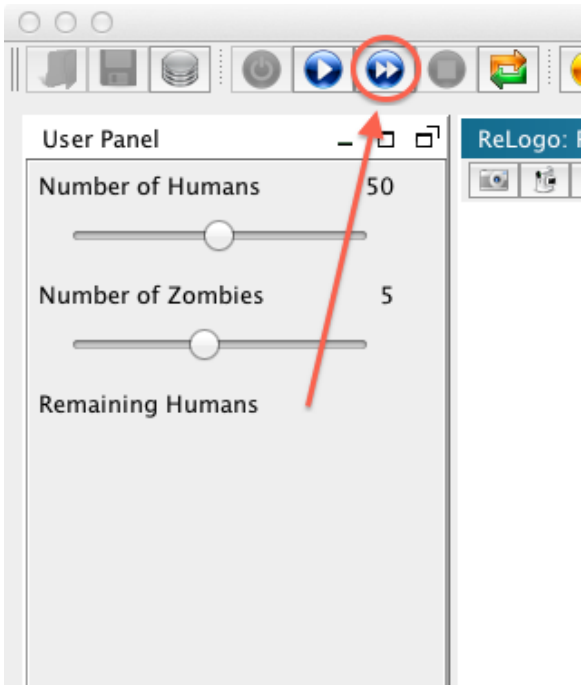

FIGURE 20. The *Step* button in the Repast Simphony runtime.

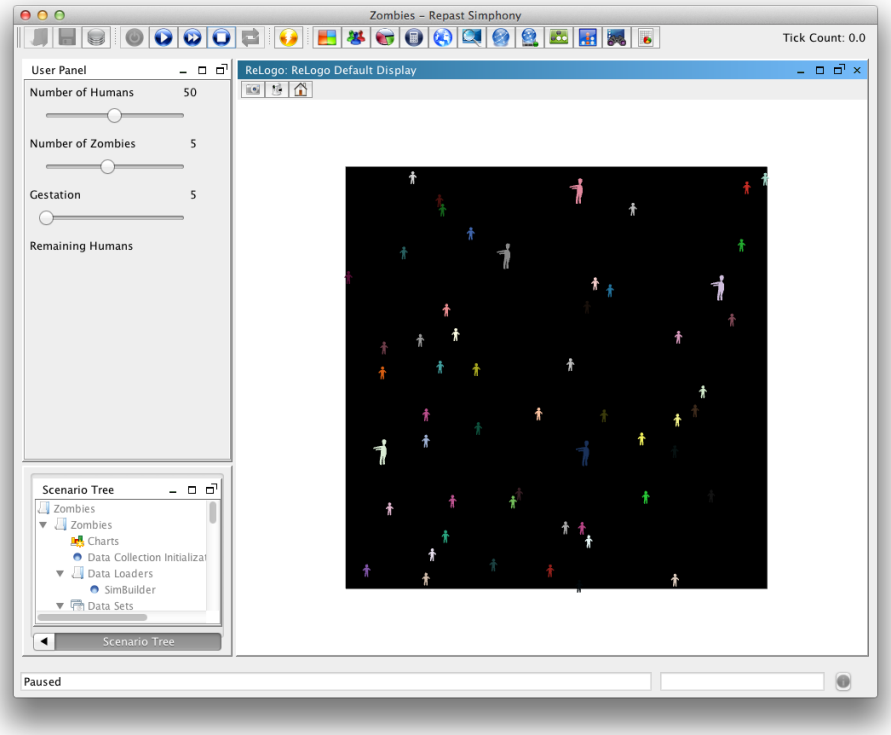

FIGURE 21. The Zombies model after pressing the *Step* button.

FIGURE 22. A closeup of the Zombies model.

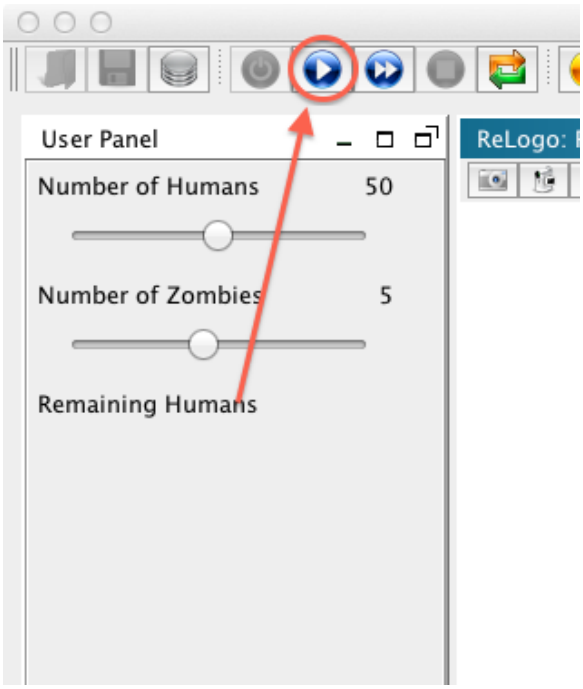

FIGURE 23. The *Play* button in the Repast Simphony runtime.

FIGURE 24. The *Pause* button in the Repast Simphony runtime.

FIGURE 25. The *Stop* button in the Repast Simphony runtime.

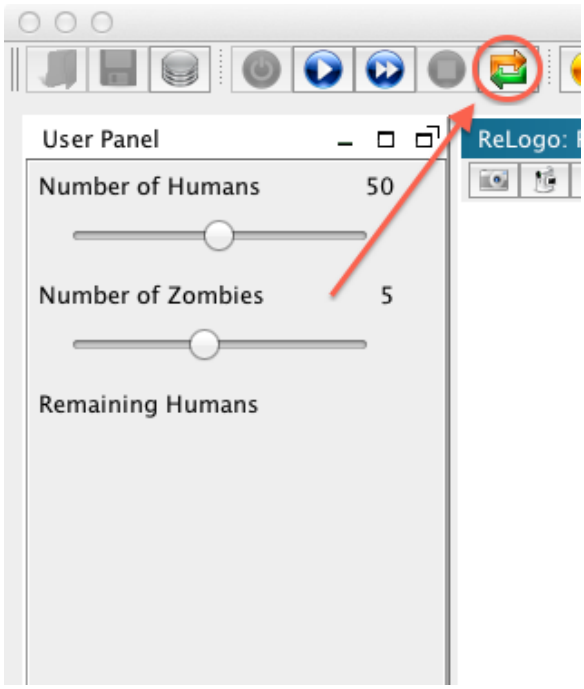

FIGURE 26. The *Reset* button in the Repast Simphony runtime.

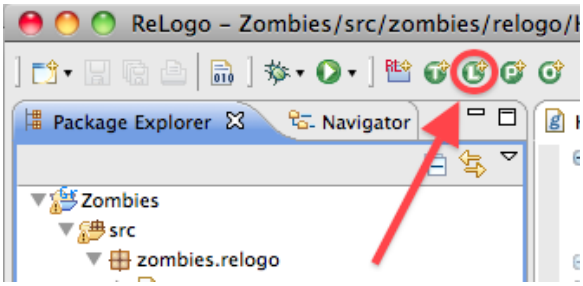

FIGURE 27. The New Link icon.

1.6. **Adding links to the Zombies model.** In this section we'll add to the Zombies model by introducing links[36]. We'll create Infection link types and use them to track which Human agents have been infected by which Zombie agent. We do this, in a similar manner to how we created our Human and Zombie turtle types, by selecting the "zombies.relogo" package and then clicking on the New Link icon (Fig. 27) in the toolbar. This brings up the New Link Wizard which allows us to specify the name of our link type. We fill in the Name field with "Infection" and press the Finish button.

We slightly modify the Zombie *step* method to look like:

[36]Links were introduced to Logo by NetLogo. They are edges of networks where the vertices are the turtles.

```
1  def step(){
2     def winner = maxOneOf(neighbors()){
3        count(humansOn(it))
4     }
5     face(winner)
6     forward(0.5)
7     if (count(humansHere()) > 0){
8        label = "Brains!"
9        def infectee = oneOf(humansHere())
10       infect(infectee)
11       createInfectionTo(infectee)
12    }
13    else {
14       label = ""
15    }
16 }
```

LISTING 10. New Zombie step method with lines 9 - 11 modified from the previous Zombie step (cf. line 15 in Listing 3).

We've made it so that if the Zombie finds a Human, they not only infect the Human but also create an Infection link to it (lines 9 - 11 in Listing 10). When the infectee dies any links it has are automatically removed so the newly hatched Zombie won't retain those[37]. In an analogous manner to how turtle type specific methods were generated when we defined new turtle types, when we create new link types we gain access to a number of new link specific methods. *createInfectionTo* is such a method. To see a full list of the methods generated see Table 2 in Appendix A.

We'd also like to be able to play with the length of the gestation period, i.e., the period it takes for a Human to die and become a Zombie, to be able to see a network of infections forming before the infected Human dies. We add a gestationPeriod variable as a slider (Listing 11) and modify the Human turtle type (Line 17 in Listing 12) to accomplish this.

[37]It is left to the interested reader to figure out how to make these Infection links persist. There are many possible ways to accomplish this but one is to use the primitives *inInfectionNeighbors* and *createInfectionsFrom*.

```
1  // package declaration and imports, which we can ignore for now
2
3  public class UserGlobalsAndPanelFactory
4  extends AbstractReLogoGlobalsAndPanelFactory{
5
6     public void addGlobalsAndPanelComponents(){
7
8        /**
9         * Example comments
10        */
11
12       addSliderWL("numHumans", "Number of Humans", 1, 1, 100, 50)
13       addSliderWL("numZombies", "Number of Zombies", 1, 1, 10, 5)
14       addSliderWL("gestationPeriod", "Gestation", 5, 1, 30, 5)
15       addMonitorWL("remainingHumans", "Remaining Humans", 5)
16    }
17
18 }
```

LISTING 11. The UserGlobalsAndPanelFactory class with a gestationPeriod variable as a slider.

```
1  // package declaration and imports, which we can ignore for now
2
3  class Human extends BaseTurtle {
4
5     def infected = false
6     def infectionTime = 0
7
8     def step(){
9        def winner = minOneOf(neighbors()){
10          count(zombiesOn(it))
11       }
12       face(winner)
13       forward(1.5)
14
15       if (infected){
16          infectionTime++
17          if (infectionTime >= gestationPeriod){
18             hatchZombies(1){
19                size = 2
20             }
21             die()
22          }
23       }
24    }
25
26 }
```

LISTING 12. The Human turtle type modified to use a variable gestation period.

Now when we run our model, we can see larger networks of Infection links forming as we increase the gestation period (Fig. 28).

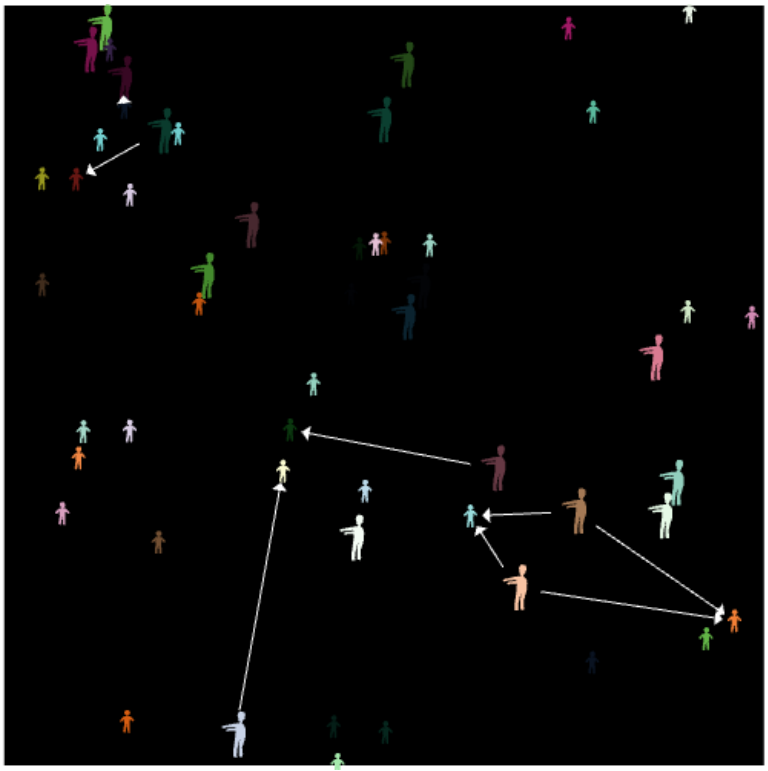

FIGURE 28. The Zombie model showing infection networks.

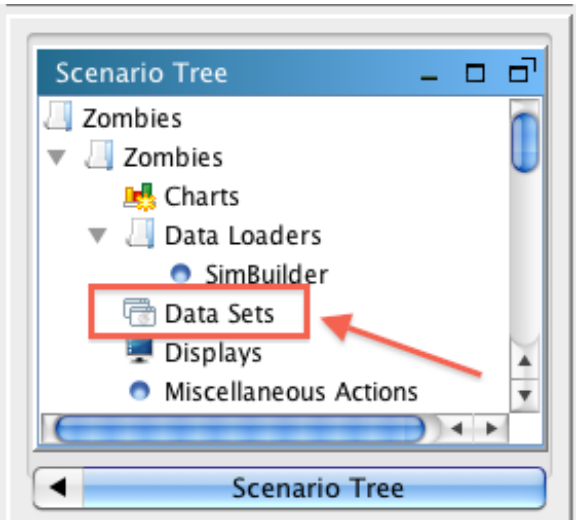

FIGURE 29. The Data Set label in the Scenario Tree panel.

1.7. **Data sets, outputters and external plugins.** Now that we have the model running with turtles, patches and links, we'll explore the creation of simple data sets to create file outputters and even to take advantage of the many plugins available which connect to external tools. We launch the Zombies model, if it's not already launched. Once it is up, we go to the Scenario Tree panel underneath the User Panel and carry out the following steps:

(1) Right click on the Data Sets label (Fig. 29) and choose Add DataSet.
(2) In the Data Set Editor, type Agent Counts as the Data Set ID, and Aggregate as the Data Set type. Click Next.
(3) In the next few steps, we specify the data sources that make up our data set. The Standard Sources tab allows us to add some standard data sources to our data set. For this tutorial we keep the default settings (only the *Tick Count* box checked).
(4) Next, select the Method Data Sources tab. The Method Data Sources tab allows us to create data sources that will call methods on agents in our model. Click on the *Add* button to add a row. Type in "Remaining Humans" for the *Source Name*, select Human for the *Agent Type* and *Count* for the *Aggregate Operation*.
(5) Click Add again. This time type in "Remaining Zombies" for the *Source Name*, select Zombie for the *Agent Type* and once again *Count* for the *Aggregate Operation*.

At this point you should see the Data Set Editor window looking like Fig. 30. Click on Next, then on Finish. Don't forget to save this setting by clicking on the save icon in the top left hand side of the Repast Simphony runtime (Fig. 31). Alternatively select Save under the File menu.

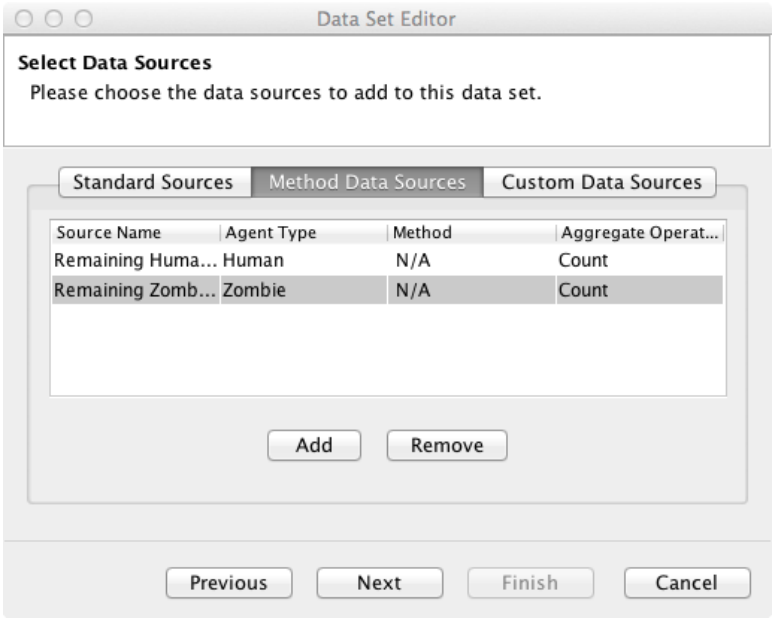

FIGURE 30. The Data Set Editor window.

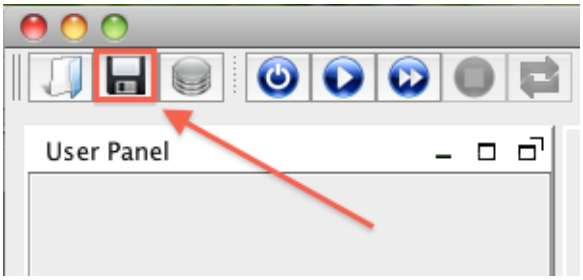

FIGURE 31. The Save icon in the Repast Simphony runtime toolbar.

Now that we've created a data set we want to be able to output this data set. We have the option of outputting the data set to the console or, as we'll be doing next, to output the data to a file sink. To do this we carry out the following steps:

(1) Right click on Text Sinks in the Scenario Tree (you may have to scroll down a bit), and click Add File Sink.
(2) Choose Agent Counts for the Data Set ID.

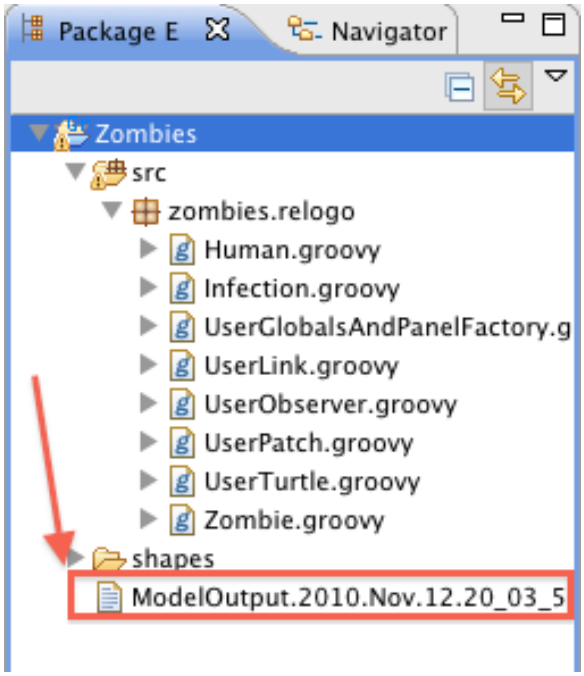

FIGURE 32. The outputted data in the Zombies model.

(3) Click on *tick* in the first column and click the green right arrow.
(4) Repeat for the remaining items in the column.
(5) Click Next, then Finish (the default location for the text sink is good for the purpose of this guide).

We've chosen the data items from our data set that we wish to output, in this case in comma-delimited form. As we did for the data set, don't forget to save this setting (Fig. 31).

Now we initialize the runtime (Fig. 17) and run the model via the *Step* or *Play* buttons. At this point if we quit out of the runtime and return to our workspace we may notice that nothing has changed. However if we select our Zombies project and "Refresh" it[38], either by right clicking on the project and choosing Refresh from the drop down menu or by pressing the F5 button, we should see the newly generated data file (Fig. 32). Double-clicking on this file will reveal a comma-delimited four column data set.

We can also directly connect to some external tools. To demonstrate this, we launch the Zombies model again. After initializing the runtime, and running the model, we turn our attention to the External Tools buttons at the top of the runtime window (Fig. 33).

[38]The Refresh is necessary when outside processes, in this case our running model, change files from outside the workspace.

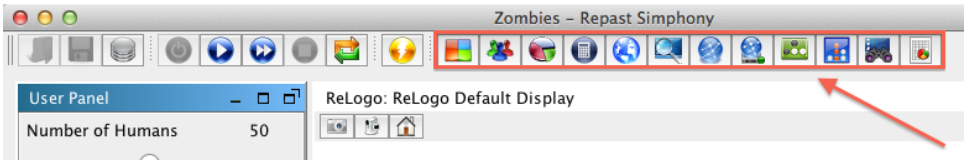

FIGURE 33. The External Tools buttons in the Repast Simphony runtime.

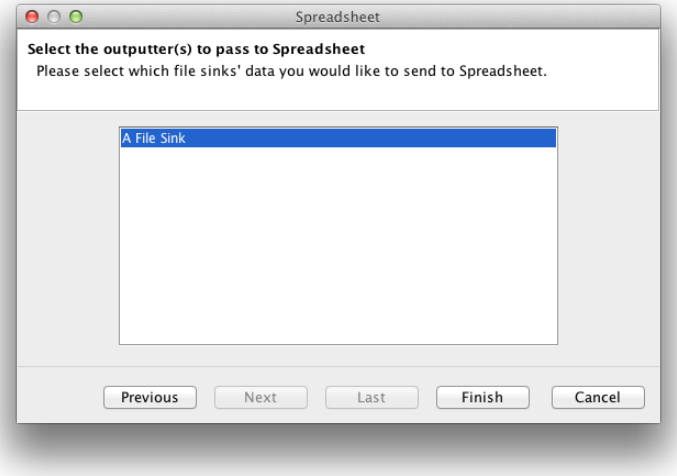

FIGURE 34. The Spreadsheet wizard with the file sink selected.

For now we'll demonstrate how we can easily export our data into Microsoft Excel. To do this we choose the Spreadsheets plugin, the button with the calculator icon. If you have Excel on your machine, chances are the default location is correct. Otherwise select the appropriate location via the Browse button. Click Next and you should see that the file sink we defined is selected (Fig. 34). Click Finish and Excel should launch with the data displayed in a spreadsheet. We recommend experimenting with the various other external tool plugins on your own.

2. A little more ReLogo.

We've gone through many of the basics for building models in ReLogo. While there are many more details that are beyond the scope of this getting started guide, below we include a few important items that we haven't covered yet.

2.1. **@Diffusible variables in patches.**

Patches can have variables with the @Diffusible annotation:

```
@Diffusible
def aPatchVariable
```

Patch variables annotated this way can be manipulated over all the patches simultaneously with the observer primitives:

- diffuse(String, double)
- diffuse4(String, double)
- diffusibleAdd(String, Number)
- diffusibleApply(String, DoubleFunction)
- diffusibleDivide(String, Number)
- diffusibleMultiply(String, Number)
- diffusibleSubtract(String, Number)

2.2. **@Plural annotation for turtles and links.**

The default plural form for turtle and link types is an 's' appended to the class name. Turtle and link types with unusual plural forms can have @Plural annotations specifying custom pluralizations.

```
@Plural("Mice")
class Mouse extends BaseTurtle {
...
}
```

This affects the names of the generated methods in Table 1 and Table 2. The custom plural forms replace the simple plural forms.

2.3. **@Undirected/@Directed annotations for links.**

By default link types are directed. To specify that a link type is undirected, or even to be explicit about a link type being directed, we can use the @Undirected and @Directed annotations on a link class.

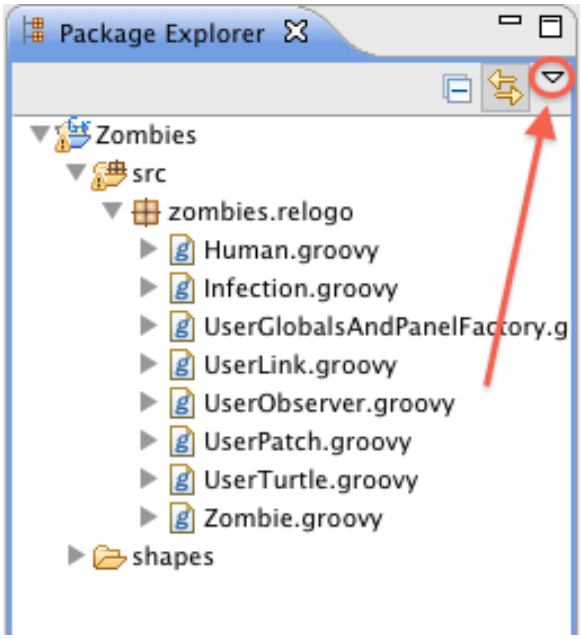

FIGURE 35. The Package Explorer's view options down arrow.

```
@Undirected
class Infection extends BaseLink {
...
}
```

2.4. **ReLogo Resource Filter.** By default the Package Explorer has a ReLogo Resource Filter selected. When in the ReLogo Perspective this hides many of the non-immediately-essential elements in a user project. At some point, however, it will be necessary to deselect this filter to access some of the hidden resources. To do this, select the Package Explorer's view options down arrow (Fig. 35) and navigate to the *Filters...* element (Fig. 36). This will reveal a listing of the available Package Explorer filters. Find and deselect the ReLogo Resource Filter here. This will reveal the previously hidden resources. Once you've gone through this process, the ReLogo Resource Filter will be visible in the view options menu (Fig. 37). At this point reenabling (or disabling) the filter can be done directly through the menu.

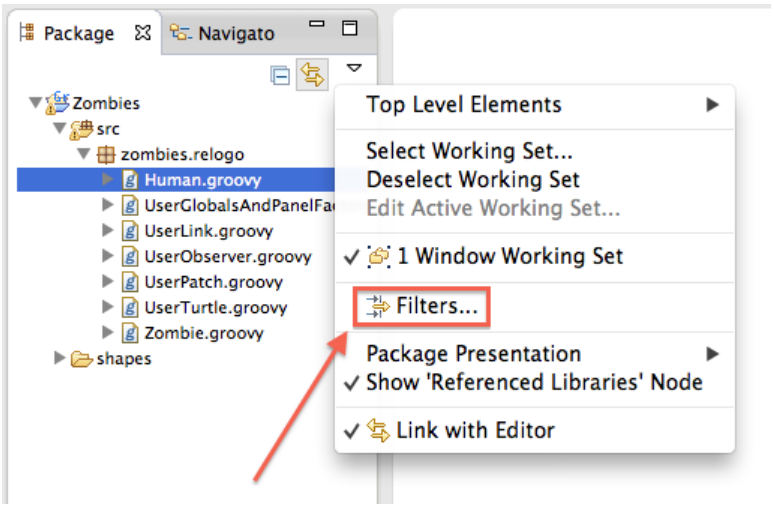

FIGURE 36. The drop down menu for the Package Explorer view options.

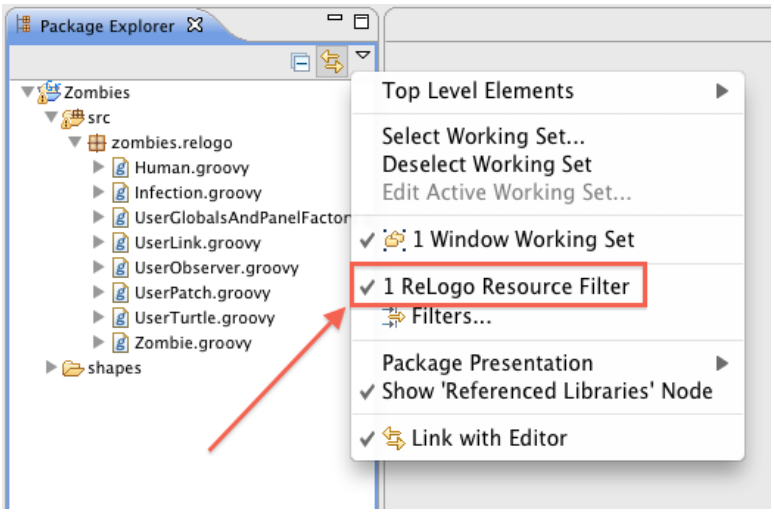

FIGURE 37. The drop down menu for the Package Explorer view options after the ReLogo Resource Filter has been modified.

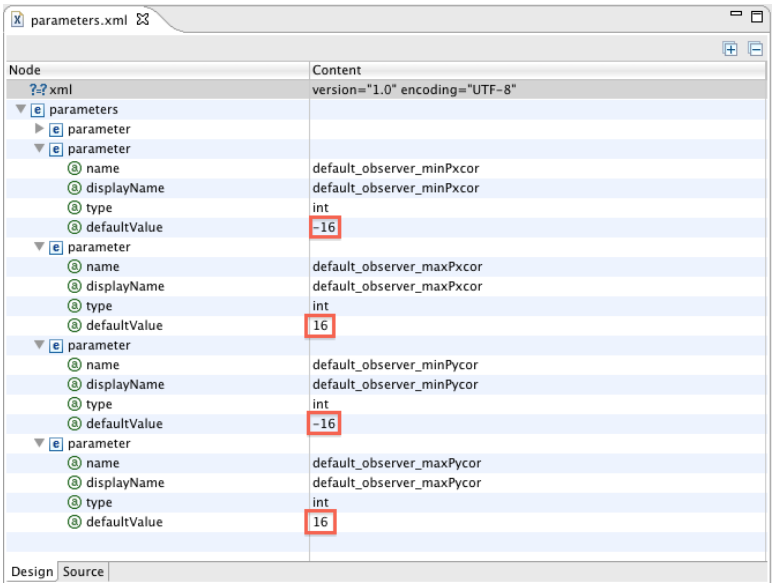

FIGURE 38. The parameters.xml file with ReLogo world dimensions highlighted.

2.5. **ReLogo world dimensions.** After disabling the ReLogo Resource Filter (Section 2.4), one of the elements that can be accessed is the parameters.xml file within the Zombies.rs folder. This is where the world dimensions are stored[39]. Double-clicking the file will display the contents in an XML editor (Fig. 38) where they can be modified. Note that the dimensions are integers and "minPxcor" and "minPycor" should be less than or equal to 0.

[39]Other parameters can be defined here as well. See the Repast Simphony documentation for details on creating and using custom parameters in your model.

3. Parameter Sweeps and Model Distribution in Repast Simphony

3.1. **Stochastic Execution and Parameter Sweeps.** Most Repast models use random draws and are therefore stochastic simulations. Stochastic simulations will produce different outcomes for different random number streams, which are generally driven by choosing different random seeds. Such simulations should be executed repeatedly to explore the space of possible outcomes. Even without randomness, model sensitivity analysis parameter sweeps should be run to determine the response of the model to changes in input values. Repast provides both local and distributed tools for automatically completing stochastic execution runs and parameter sweeps. Please see the Repast Parameter Sweep Guide for more information.

3.2. **Model Distribution.** Repast models can be distributed to model users via the installation builder. This feature packs up your model and all of the software you need to run it, except for a properly configured Java Runtime Environment, into a single Java archive ("JAR") file that can be given to model users. The resulting installer can be executed on any system with Java version 7 or later installed. Users simply copy the installer file onto their Windows, Mac OS, or Linux computers and the start the installer by double clicking on the file. Once the installer is started it will show an installation wizard that will prompt the user for the information needed to install the model. If desired, the installer can also be run in a command line mode.

Building an installer for a model is straightforward. Simply choose the "Build Installer for ⟨Your Model Name Here⟩ Model" and provide a location and name for the installer file. The installer file's default name is "setup.jar," which is suitable for most purposes. The install builder will then package and compress your model and the supporting Repast software. The resulting installer files are about 70 MB plus the size of the model code and data. 75 MB to 80 MB is a common total size.

The Repast install builder uses the IzPack system (http://izpack.org/). More information on installer customization and use, including command line activation, can be found on the IzPack web site.

Appendix A. Generated ReLogo Primitives

When a new turtle type is defined a number of methods become available to turtles, patches, links and observers. These methods are added to the ReLogoTurtle, ReLogoPatch, ReLogoLink, and ReLogoObserver classes in the project's `src-gen` directory[40]. All user defined turtle, patch, link and observer classes extend these classes and therefore inherit their methods. The specific methods generated when a Zombie turtle type is defined are summarized in Table 1. These are similar in functionality to existing ReLogo primitives with similar names but specialized for the particular turtle type. For more information on the generated methods you can make use of the informational pane available in the ReLogo editor when hovering over any of them or you can navigate to the `src-gen` source folder directly.

Table 1. Methods generated when a Zombie turtle type is defined.

Generated for each			
turtle	patch	link	observer
hatchZombies	sproutZombies		
zombiesHere	zombiesHere		
zombiesAt	zombiesAt		
zombiesOn	zombiesOn	zombiesOn	zombiesOn
isZombieQ	isZombieQ	isZombieQ	isZombieQ
zombies	zombies	zombies	zombies
zombie	zombie	zombie	zombie
			createZombies
			createOrderedZombies

When a new link type is defined methods are generated and become available to turtles, patches, links and observers. The specific methods created when a Connection link type are listed in Table 2. Note that there are differences depending on whether Connection is a directed or undirected link type. Again, these are similar in functionality to existing ReLogo primitives with similar names but specialized for the particular link type.

[40]The `src-gen` source directory can be made visible by disabling the ReLogo Resource Filter (Section 2.4).

TABLE 2. Methods generated when a Connection link type is defined.

Generated for each[a] [b]			
turtle	patch	link	observer
isConnectionQ	isConnectionQ	isConnectionQ	isConnectionQ
connections	connections	connections	connections
connection	connection	connection	connection
createConnectionFrom[a]			
createConnectionsFrom[a]			
createConnectionTo[a]			
createConnectionsTo[a]			
createConnectionWith[b]			
createConnectionsWith[b]			
inConnectionNeighborQ[a]			
inConnectionNeighbors[a]			
inConnectionFrom[a]			
myInConnections[a]			
myOutConnections[a]			
outConnectionNeighborQ[a]			
outConnectionNeighbors[a]			
outConnectionTo[a]			
connectionNeighborQ			
connectionNeighbors			
connectionWith[b]			
myConnections			

[a]For a directed link Connection.

[b]For an undirected link Connection.

Appendix B. Available Graphical Elements

Table 3 lists the various elements available for use in the UserGlobalsAndPanelFactory's *addGlobalsAndPanelComponents* method.

Table 3. Elements available for use in the UserGlobalsAndPanelFactory's *addGlobalsAndPanelComponents* method

Element Types	Commands[a]
Slider	addSlider(WL)
Chooser	addChooser(WL)
Switch	addSwitch(WL)
State Change Button[b]	addStateChangeButton(WL)[c]
Input	addInput
Monitor	addMonitor [c]
Global	addGlobal

[a]The (WL) refers to variants *with labels*.
[b]State change buttons do not advance the simulation schedule.
[c]Include variants with observer specified by id.

Additionally, *Observer*s are provided with a *registerModelParameterListener(String,Closure)* method which allows for registering a closure to execute when a specified model parameter (such as those defined by the Global, Slider, Chooser, Switch and Input elements) are modified. This method would typically be called from within a *setup*-like *Observer* method.

Groovy's SwingBuilder can also be used to create more advanced GUIs. The elements available are listed in Table 4. Listing 13 shows a code snippet example. Note that the usage pattern of the elements in Table 4 is to assign the elements to a variable and use SwingBuilder's *widget* method to incorporate them into panels you create with *addPanel*[41].

```
1 def slider = sliderWL("sliderVar", "My Variable", 1, 1, 100, 50)
2 def monitor = monitorWL("monitorMethod", "My Monitor", 5)
3 addPanel{
4     gridLayout(columns:1, rows:0)
5     widget(slider)
6     widget(monitor)
7 }
```

Listing 13. Using Groovy's SwingBuilder for building GUIs.

[41]The *addPanel* method passes its closure argument to a SwingBuilder *panel* element.

TABLE 4. Elements available for use with Groovy's SwingBuilder in the UserGlobalsAndPanelFactory's *addGlobalsAndPanelComponents* method

Element Types	Commands[a]
Panel	addPanel
Slider	slider(WL)
Chooser	chooser(WL)
Switch	rSwitch(WL)
State Change Button[b]	stateChangeButton(WL)[c]
Input	input
Monitor	monitor [c]

[a]The (WL) refers to variants *with labels*.

[b]State change buttons do not advance the simulation schedule.

[c]Include variants with observer specified by id.

Appendix C. More on Turtle Shapes

Any image file that's placed in the shapes folder of a ReLogo project can be referred to by name (minus suffix) within the model. Different platforms will have some minor differences in the types of image files that are readable but most of the standard ones can be used (e.g., "jpeg", "pbm", "bmp", "jpg", "wbmp", "ppm", "png", "jp2", "pgm", "gif"). In addition to regular image files, ReLogo allows one to create svg files that can either be used as images, which we refer to as "complex" svg files, or as shapes which have specified regions with fixed colors and regions whose colors can be changed within a ReLogo simulation. The default svg files that come with any newly created ReLogo project are of the latter type, which we refer to as "simple" svg files.

We recommend the free and open source SVG editor Inkscape for creating svg images. There are some keywords that can be used to define the special ReLogo properties of an svg file you create. These are specified as a comma delimited list in the "Keywords" field of the file's Document Metadata (File → Document Metadata...):

- simple: indicates that the svg file is simple (assumed complex if not specified)
- rotate: indicates whether the icon should rotate according to the turtle's heading
- offset *value*: indicates the clockwise offset (in degrees) of the shape

To fix the color of an element within a simple svg file right click on the element and select "Object Properties." In the Description field enter "fixedColor" and the element will retain its color. All other elements in a simple svg will take on the color assigned to the turtle within a simulation.